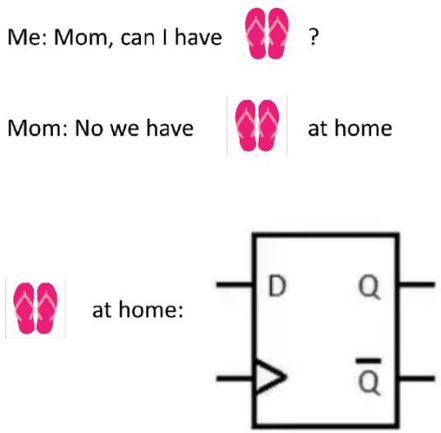
EECS 370 - Lecture 10

FSM and Single-Cycle Datapath



Announcements

- P2
  - P2a due next Thursday
  - P2L due in a month
- HW 2
  - Posted, due in ~2 weeks
- Lab 5 meets Fri/Mon
- My group office hours moved to **4941 BBB** for this week only
- Midterm Exam
  - Thu Oct 12th, 6-8 pm (2 weeks from today)
  - Sample exams on website
  - You can bring 1 sheet (double sided is fine) of notes
  - We will provide LC2K encodings + ARM cheat sheet

Agenda

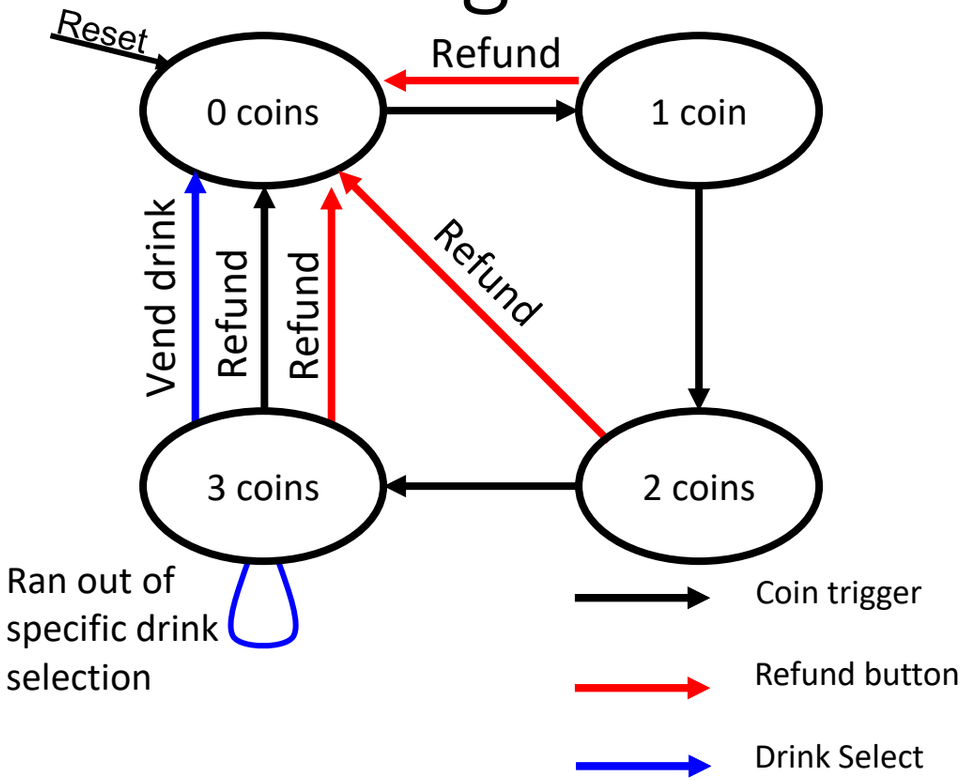
- **FSM Implementation**
- ROMs
- Making our FSM more efficient
- Single Cycle Processor Design Overview
- Supporting each instruction
  - ADD / NOR
  - LW / SW
  - BEQ
  - JALR

Finite State Machines

- Combine combinational logic and sequential logic
- Define a set of "states" that our machine will be in
- "Remember" what state we're in using flip-flops
- Calculate next-state and output logic using combinational logic

Note: This is very similar to Finite State Autonoma (FSA) from 376, but with a few differences (the input never ends, and the FSM always outputs something)

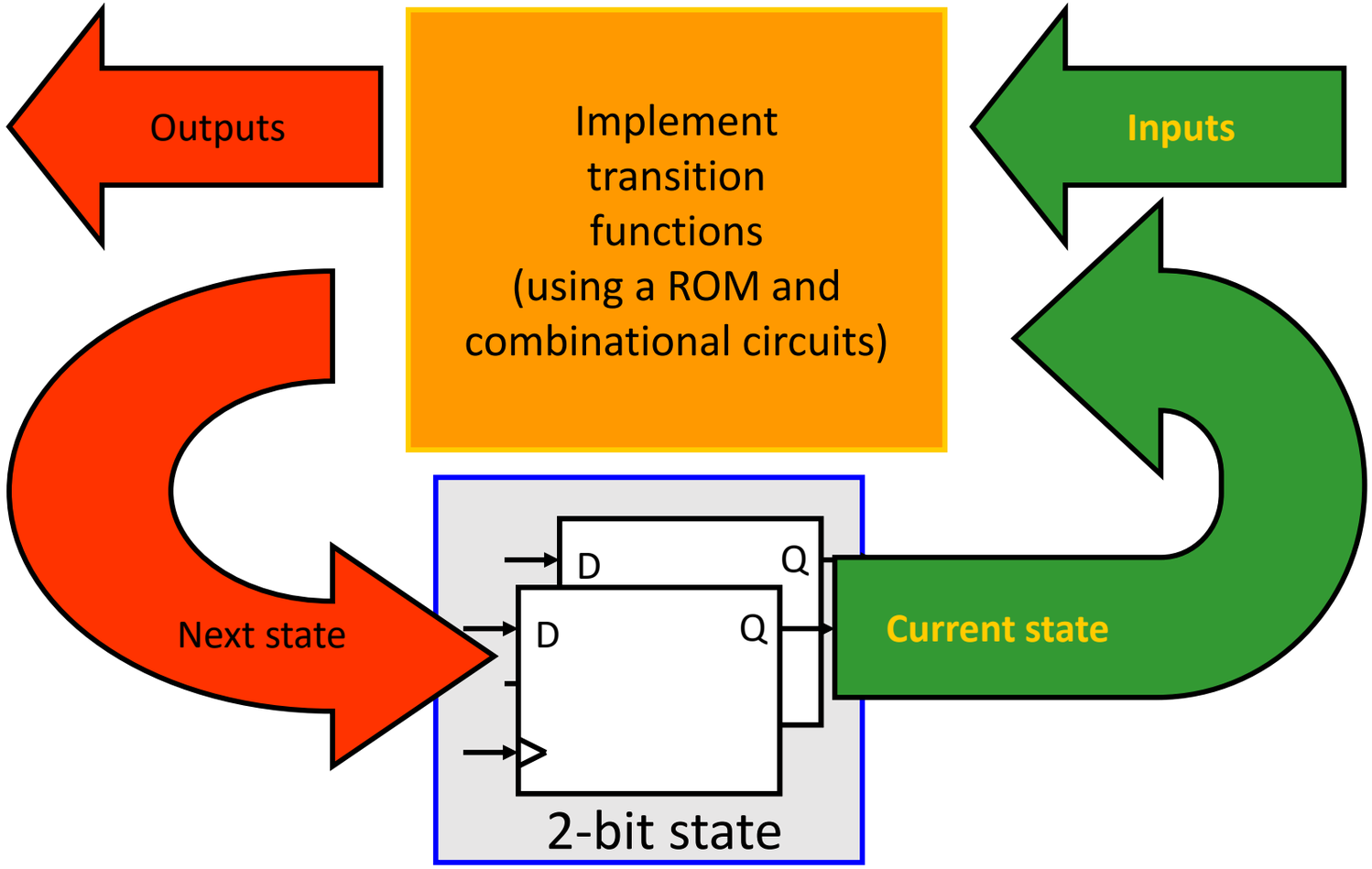
FSM for Vending Machine



Is this a Mealy or Moore Machine?

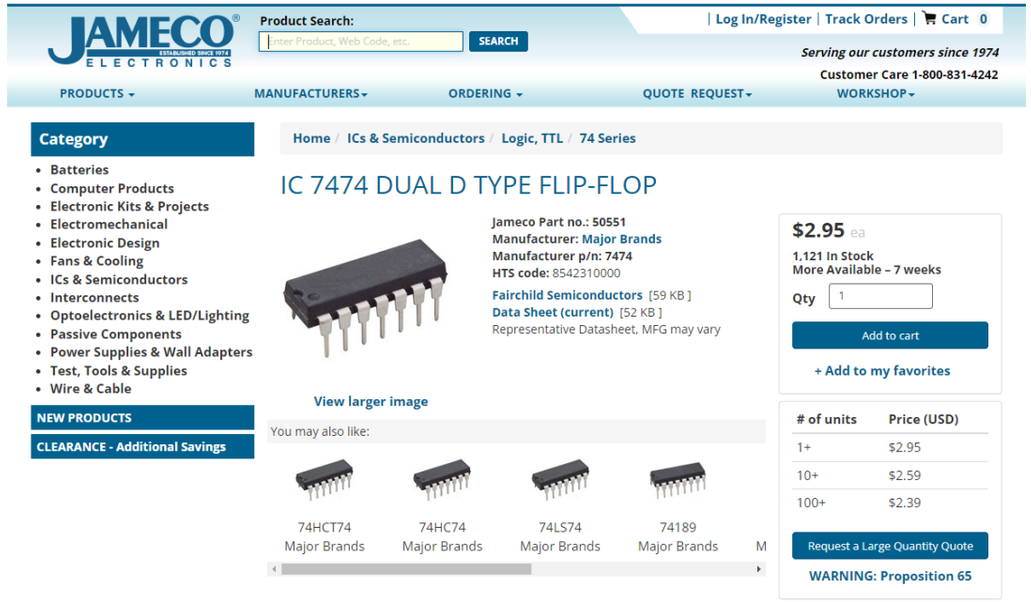
This is Mealy: Mealy output is based on current state *AND* input

Implementing an FSM



Implementing an FSM

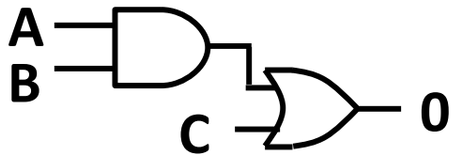
- Let's see how cheap we can build this vending machine controller!
- Jameco.com sells electronic chips we can use
  - D-Flip-flops: $3, includes several in one package
- For custom combinational circuits, would need to design and send to a fabrication facility
  - Thousands or millions of dollars!!
  - Alternative?



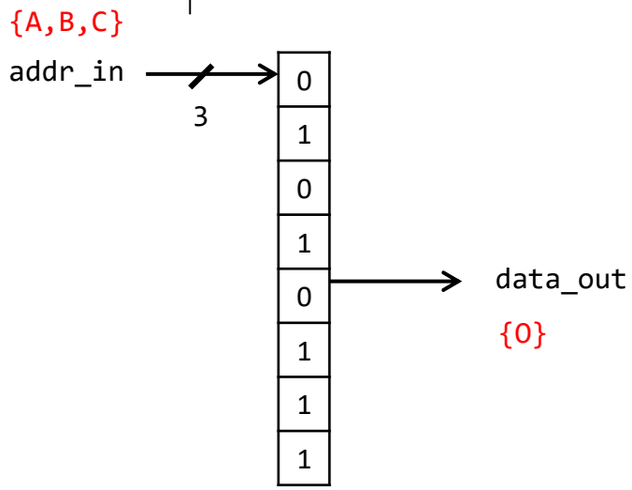
Implementing Combinational Logic

If I have a truth table:

- I can either implement this using combinational logic:



A	B	C	O
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

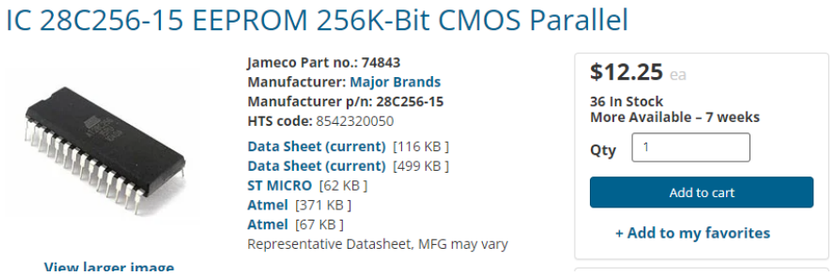


- …or I could literally just store the entire truth table in a memory and just "index" it by treating the input as a number!
  - Can be implemented cheaply using "Read Only Memories", or "ROMS"

Agenda

- FSM Implementation
- **ROMs**
- Making our FSM more efficient
- Single Cycle Processor Design Overview
- Supporting each instruction
  - ADD / NOR
  - LW / SW
  - BEQ
  - JALR

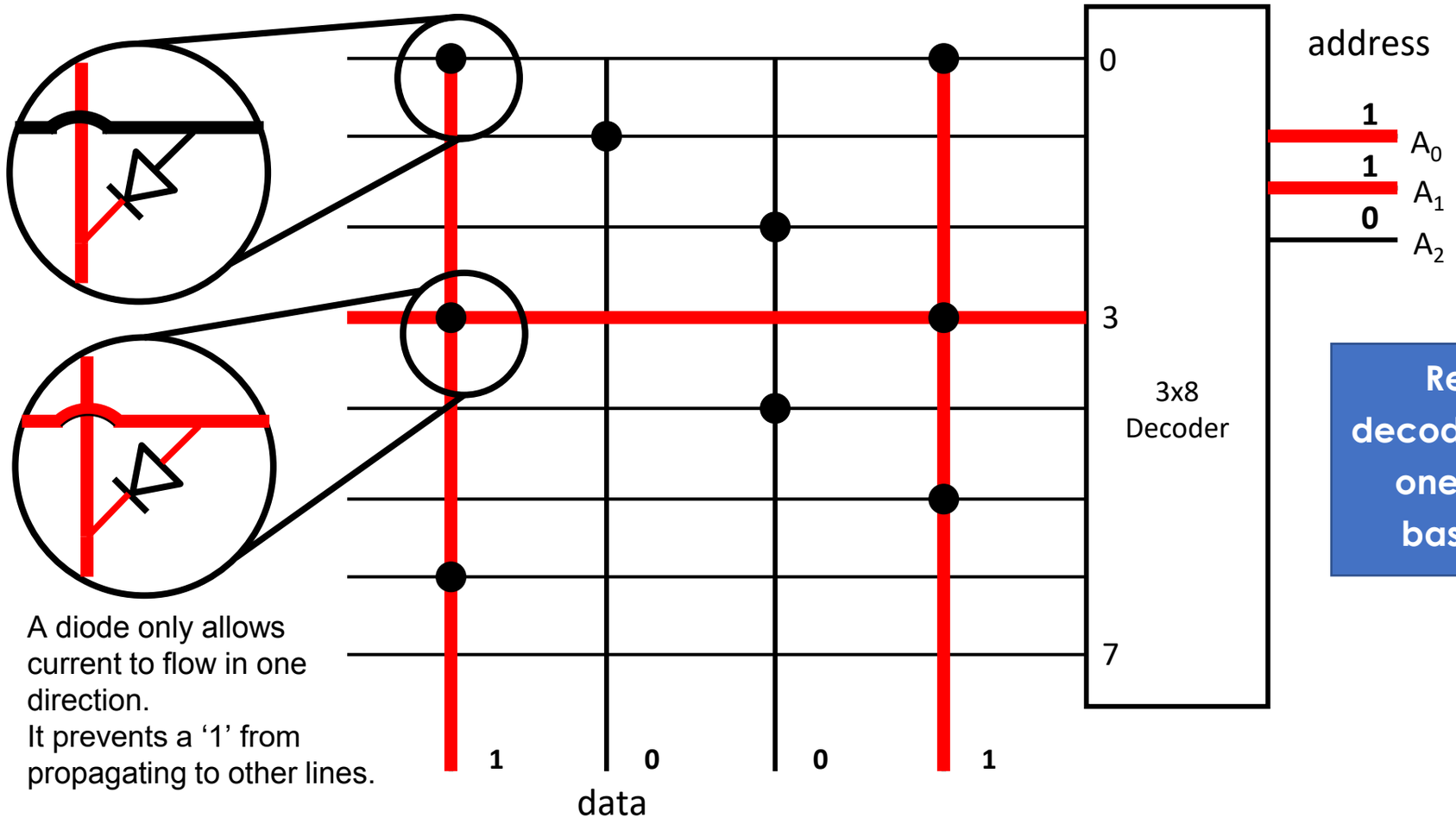
ROMs and PROMs



- Read Only Memory (ROM)
  - Array of memory values that are constant
  - Non-volatile (doesn't need constant power to save values)
- Programmable Read Only Memory
  - Array of memory values that can be written exactly once
- Electronically Erasable PROM (EEPROM)
  - Can write to memory, deploy in field
  - Use special hardware to reset bits if need to update

- 256 KBs of EEPROM costs ~$10 on Jameco
  - Much better then spending thousands on design costs unless we're gonna make **tons** of these

8-entry 4-bit ROM



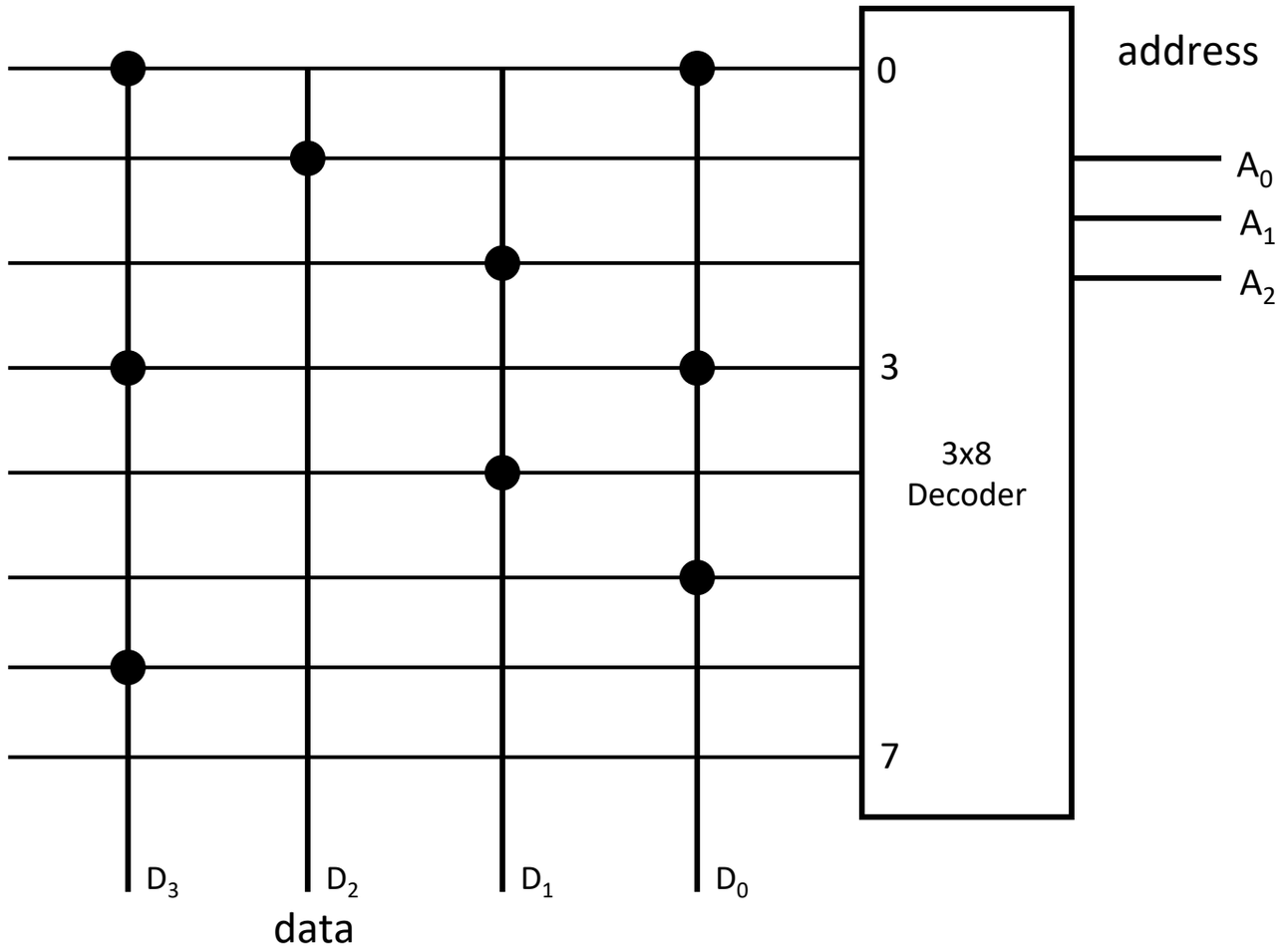
- A diode only allows current to flow in one direction.
- It prevents a ‘1’ from propagating to other lines.

Reminder: A decoder sets exactly one output high based on input

8-entry 4-bit ROM

Input	Output
000	
001	
010	
011	
100	
101	
110	
111	

This ROM corresponds to this truth table

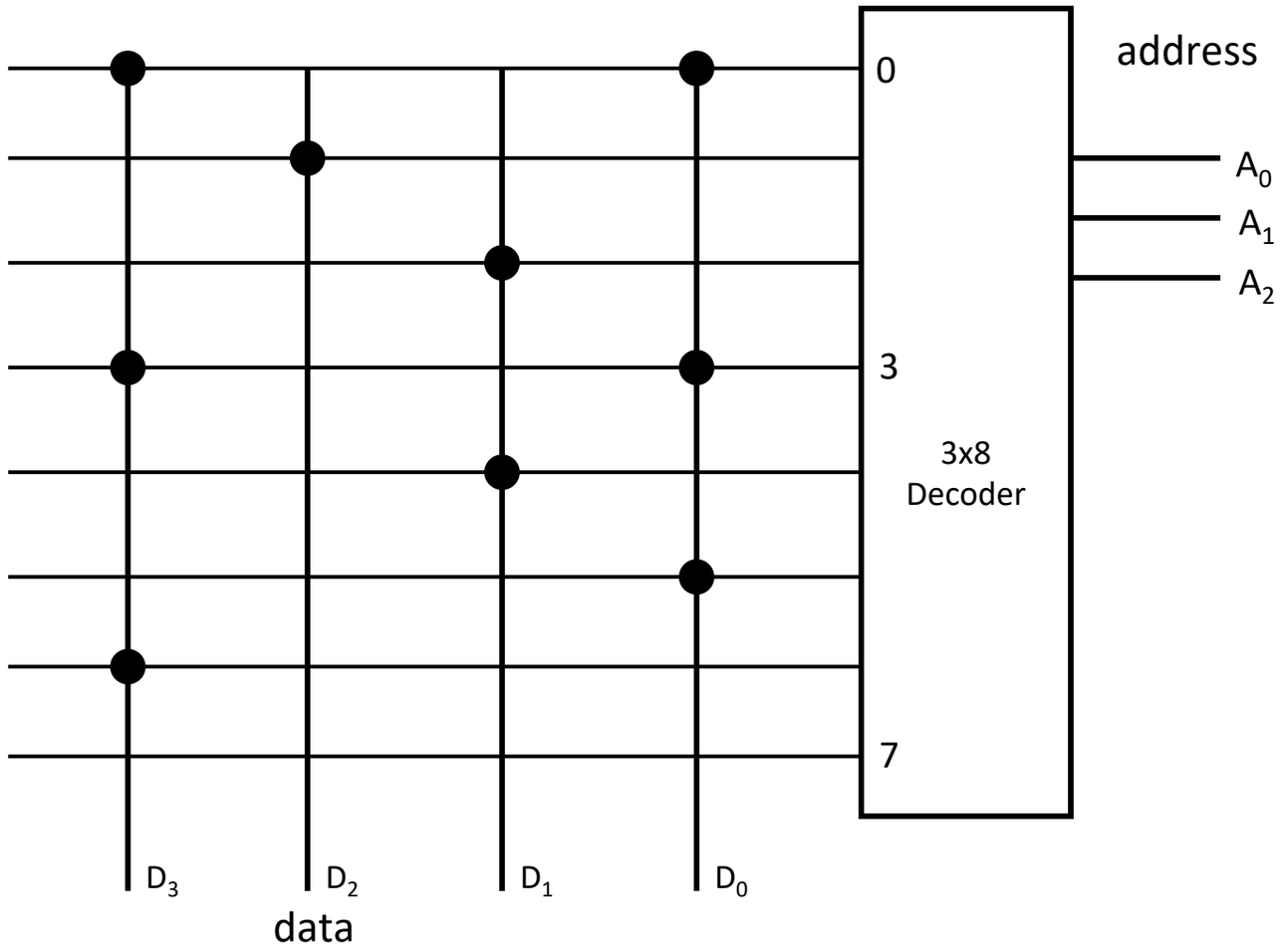


8-entry 4-bit ROM

Poll: What's the formula for size of ROM needed?

Input	Output
000	1001
001	0100
010	0010
011	1001
100	0010
101	0001
110	1000
111	0000

This ROM corresponds to this truth table



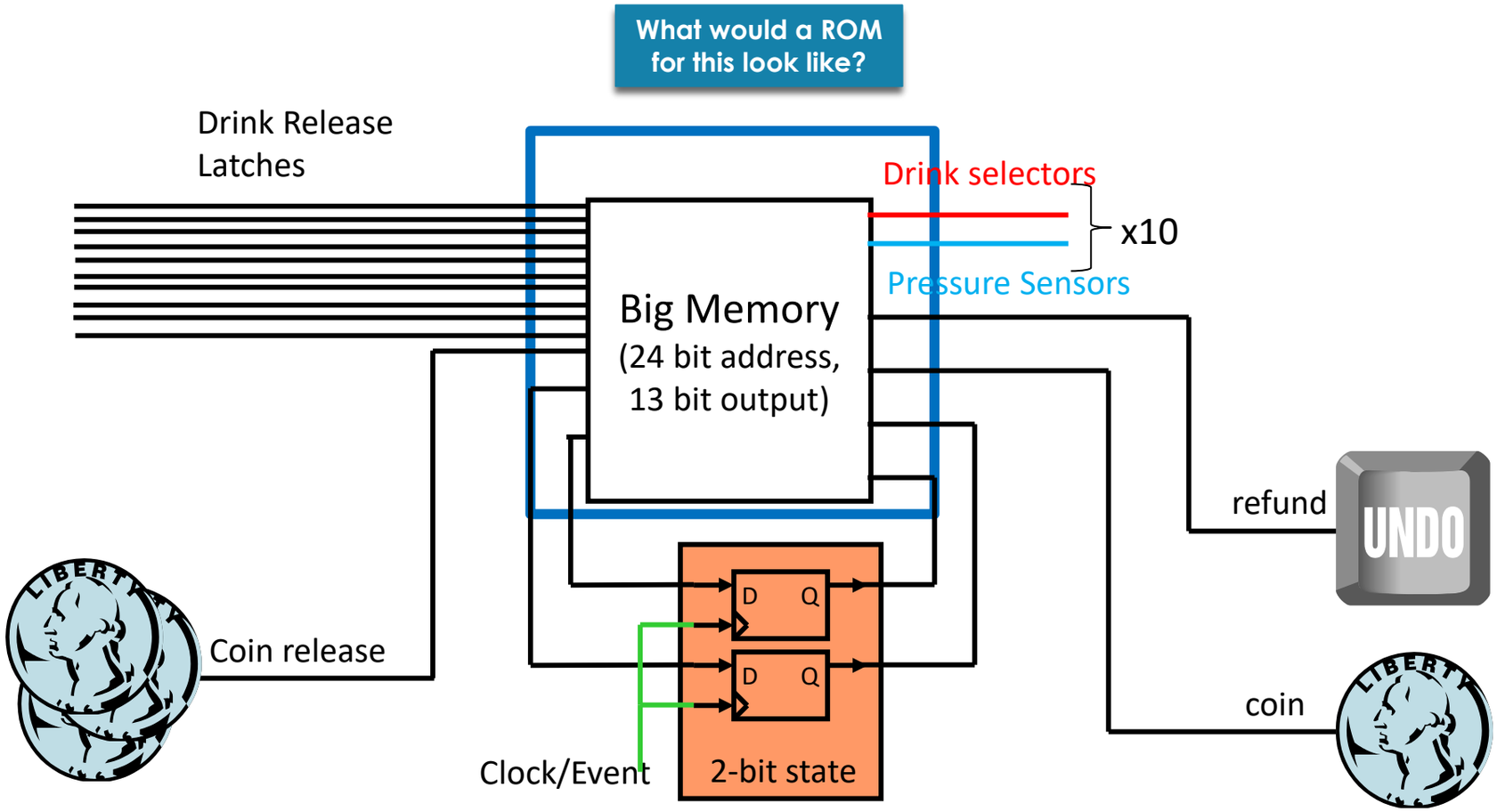
Implementing Combinational Logic

- Custom logic
  - Pros:
    - Can optimize the number of gates used
  - Cons:
    - Can be expensive / time consuming to make custom logic circuits
- Lookup table:
  - Pros:
    - Programmable ROMs (Read-Only Memories) are very cheap and can be programmed very quickly
  - Cons:
    - Size requirement grows exponentially with number of inputs (adding one just more bit **doubles** the storage requirements!)

Agenda

- FSM Implementation
- ROMs
- **Making our FSM more efficient**
- Single Cycle Processor Design Overview
- Supporting each instruction
  - ADD / NOR
  - LW / SW
  - BEQ
  - JALR

Controller Design So far



ROM for Vending Machine

Size of ROM is (# of ROM entries * size of each entry)

- # of ROM entries = $2^{input\_size} = 2^{24}$
- Size of each entry = output size = 13 bits

We need 2^{24} entry, 13 bit ROM memories

- **218,103,808 bits of ROM (26 MB)**
- Biggest ROM I could find on Jameco was 4 MB @ $6
  - Need 7 of these at $42??
- Let's see if we can do better

Reducing the ROM needed

- Idea: let's do a hybrid between combinational logic and a lookup table
  - Use basic hardware (AND / OR) gates where we can, and a ROM for everything more complicated
  - AND / OR gates are mass producible & cheap!
    - ~$0.15 each on Jameco

IC 74HC08 QUAD 2-INPUT POSITIVE AND GATE

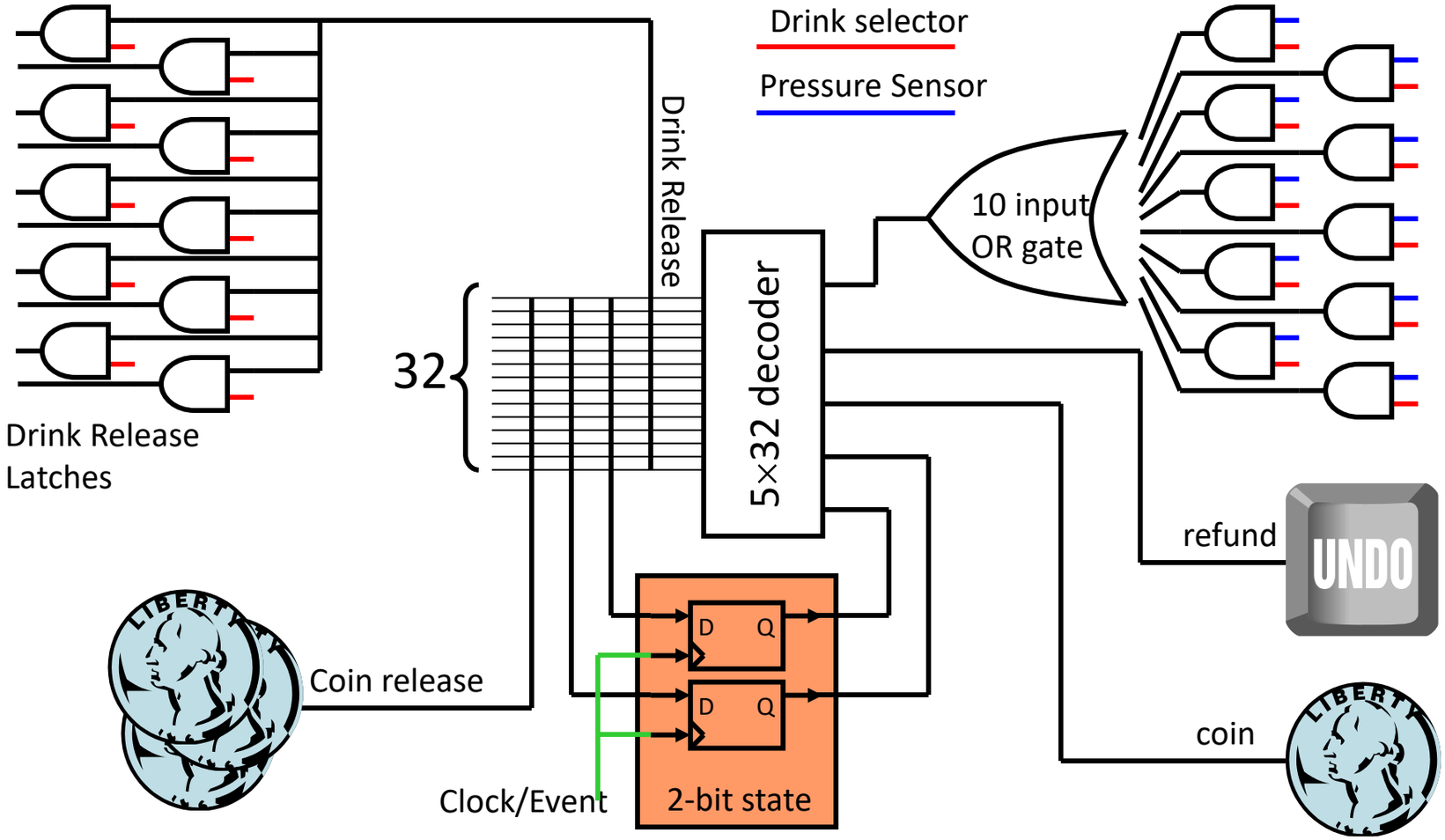
Jameco Part no.: 45225
Manufacturer: Major Brands
Manufacturer p/n: 74HC08
HTS code: 8542390000
Fairchild Semiconductors [83 KB]
Data Sheet (current) [83 KB]
Representative Datasheet, MFG may vary

$0.49 ea
1,061 In Stock
More Available – 7 weeks
Qty 1
Add to cart
+ Add to my favorites

Reducing the ROM needed

- Observation: overall logic doesn't really need to distinguish between **which** button was pressed
  - That's only relevant for choosing **which** latch is released, but overall logic is the same
- Replace 10 selector inputs and 10 pressure inputs with a **single** bit input (drink selected)
  - Use drink selection input to specify which drink release latch to activate
  - Only allow trigger if pressure sensor indicates that there is a bottle in that selection. (10 2-bit ANDs)

Putting it all together



Total cost of our controller

- Now:
  - 2 current state bits + 3 input bits (5 bit ROM address)
  - 2 next state bits + 2 control trigger bits (4 bit memory)
  - $2^5 \times 4 = 128$ bit ROM
    - 1-millionth size of our 26 MB ROM 😳
- Total cost on Jameco:
  - Flip-flops to store state: $3
  - ROM to implement logic: $3
  - AND/OR gates: $5
  - **Total:** **$11**
- Could probably do a lot cheaper if we buy in bulk

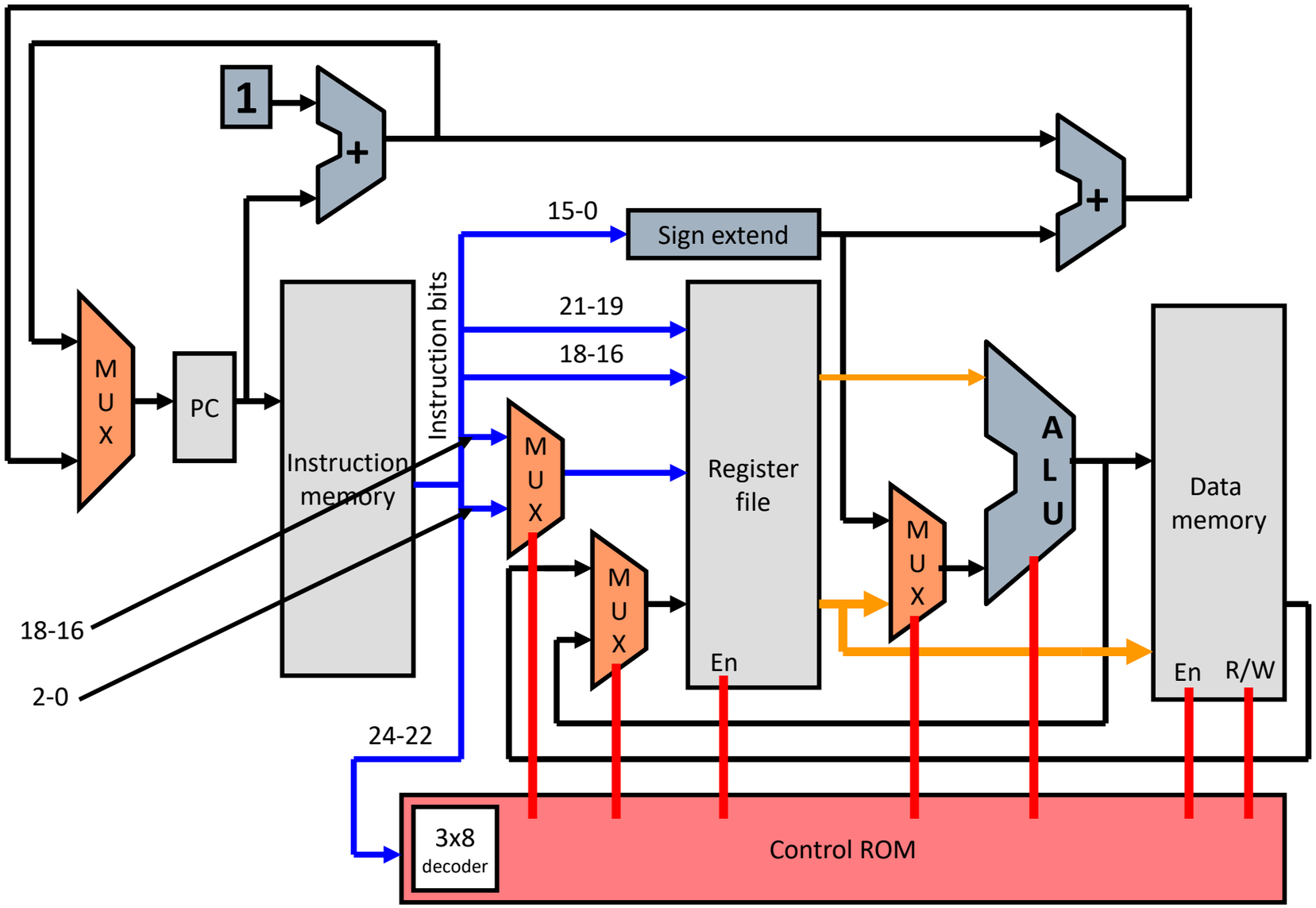
Agenda

- FSM Implementation
- ROMs
- Making our FSM more efficient
- **Single Cycle Processor Design Overview**
- Supporting each instruction
  - ADD / NOR
  - LW / SW
  - BEQ
  - JALR

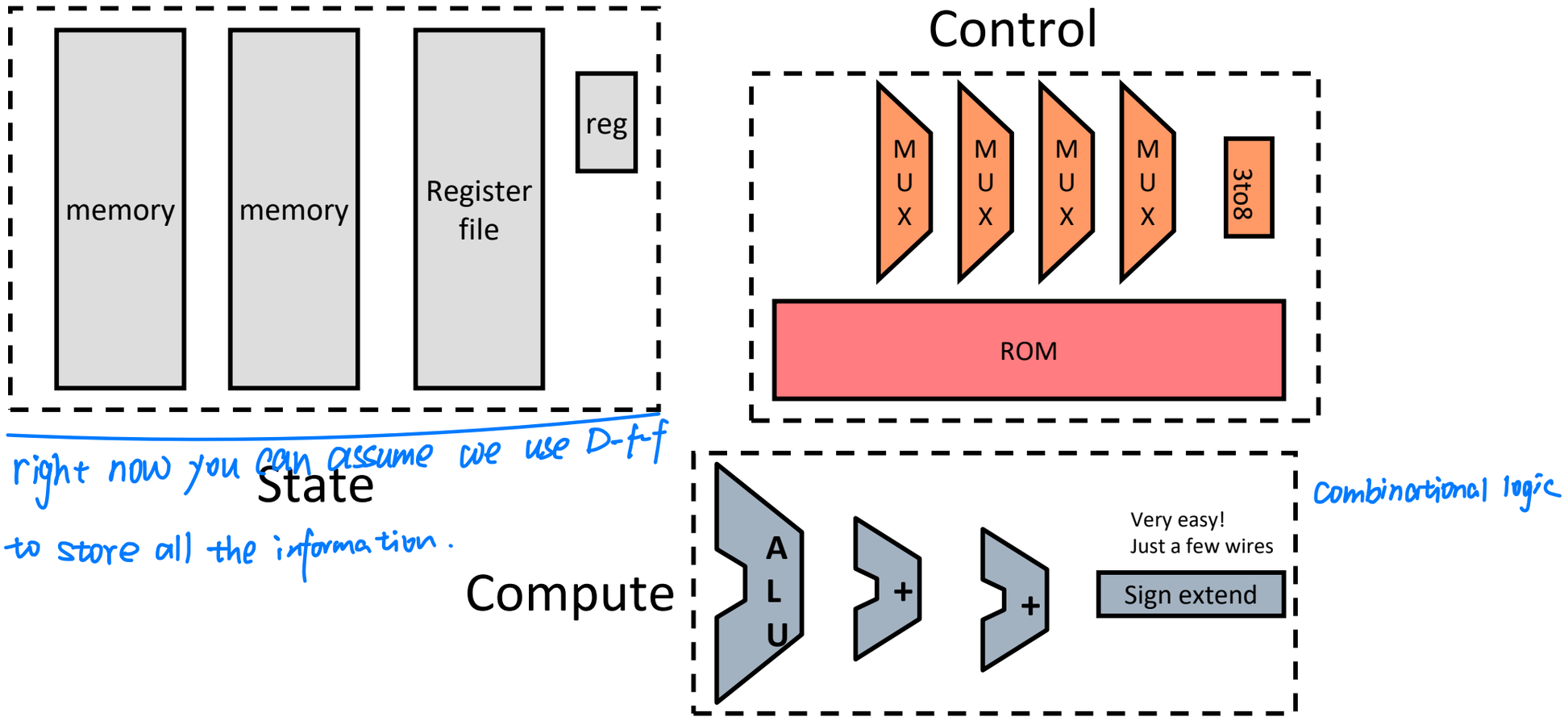
Single-Cycle Processor Design

- General-Purpose Processor Design
  - Fetch Instructions instructions are stored in the memory
  - Decode Instructions
    - Instructions are input to control ROM
  - ROM data controls movement of data
    - Incrementing PC, reading registers, ALU control
  - Clock drives it all update the state of machine. (ff)
  - Single-cycle datapath: Each instruction completes in one clock cycle

LC2K Datapath Implementation

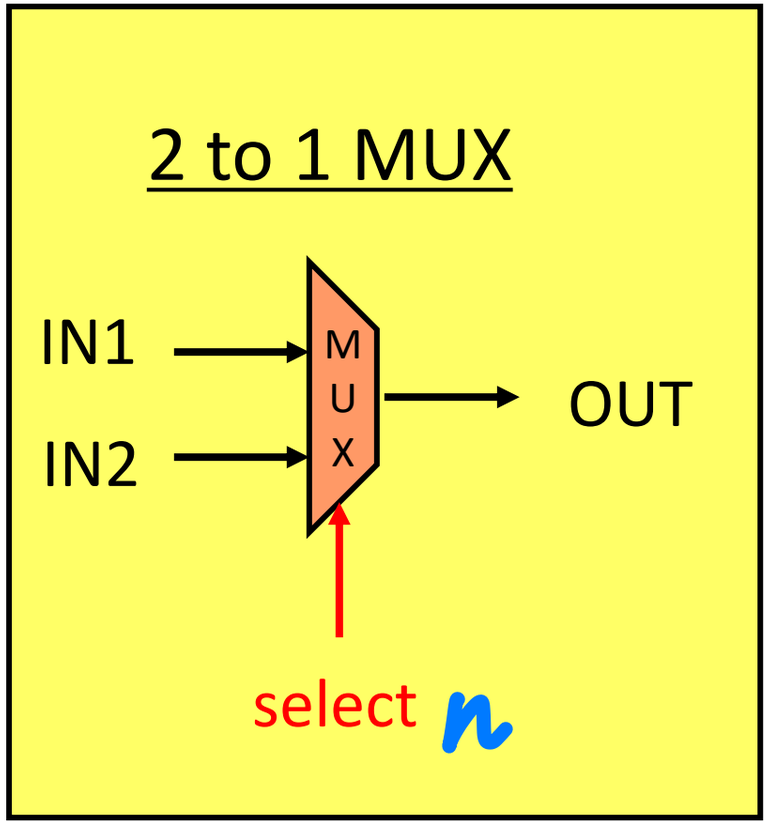


Building Blocks for the LC2K



Here are the pieces, go build yourself a processor!

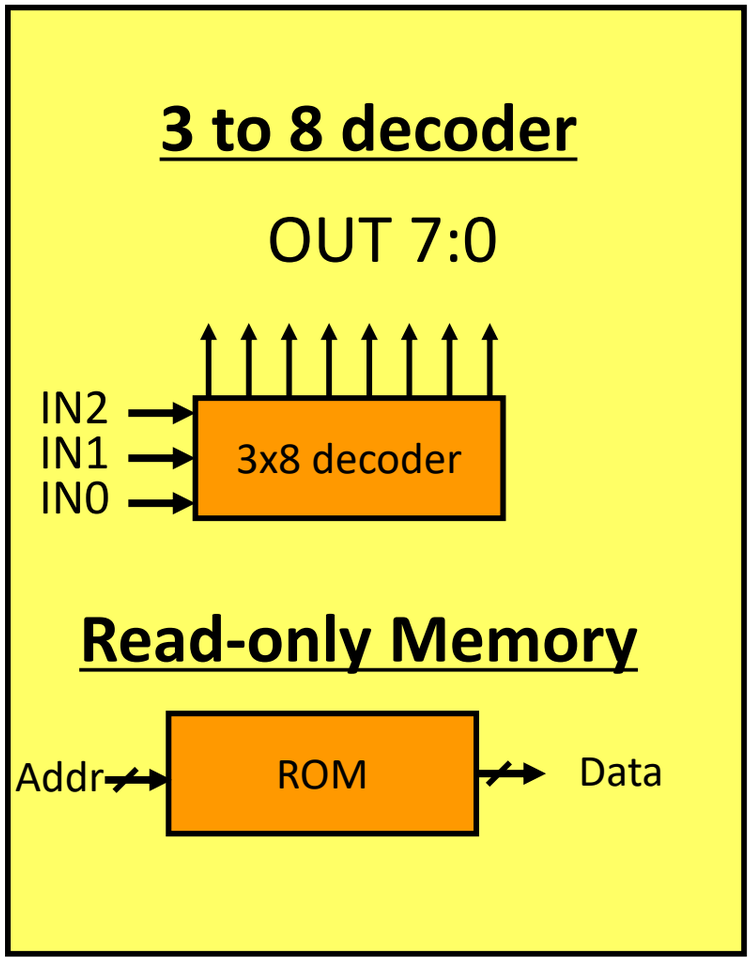
Control Building Blocks (1)



Connect one of the inputs to OUT based on the value of select

```
If (! select)
    OUT = IN1
Else
    OUT = IN2
```

Control Building Blocks (2)



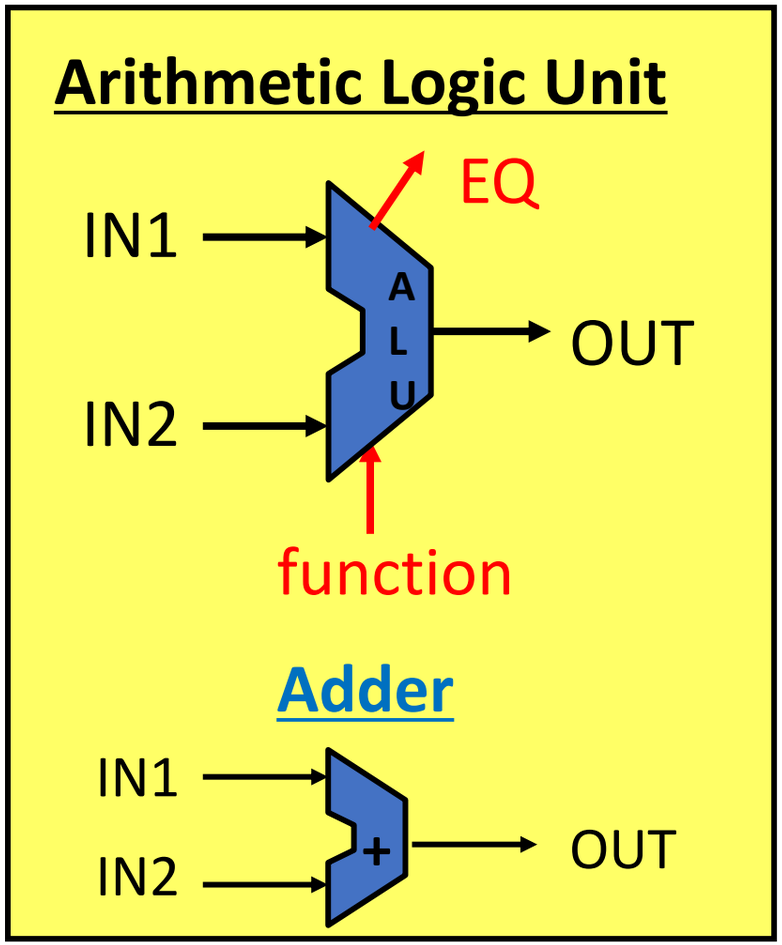
Decoder activates one of the output lines based on the input

IN	OUT
210	76543210
000	00000001
001	00000010
010	00000100
011	00001000
etc.	

ROM stores preset data in each location

- Give address, get data.

Compute Building Blocks (1)



Perform basic arithmetic functions

OUT = f(IN1, IN2)
EQ = (IN1 == IN2)

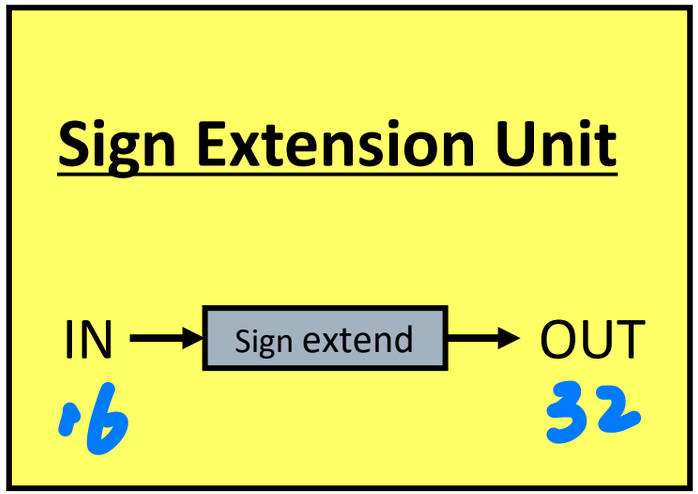
For LC2K:

f=0 is add
f=1 is nor

For other processors, there are many more functions.

Just adds

Compute Building Blocks (2)



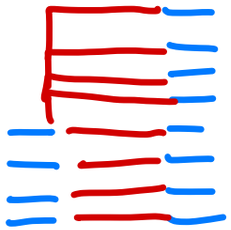
really efficient
o gate needed.

Sign extend (SE) input by replicating the MSB to width of output

OUT(31:0) = SE(IN(15:0))

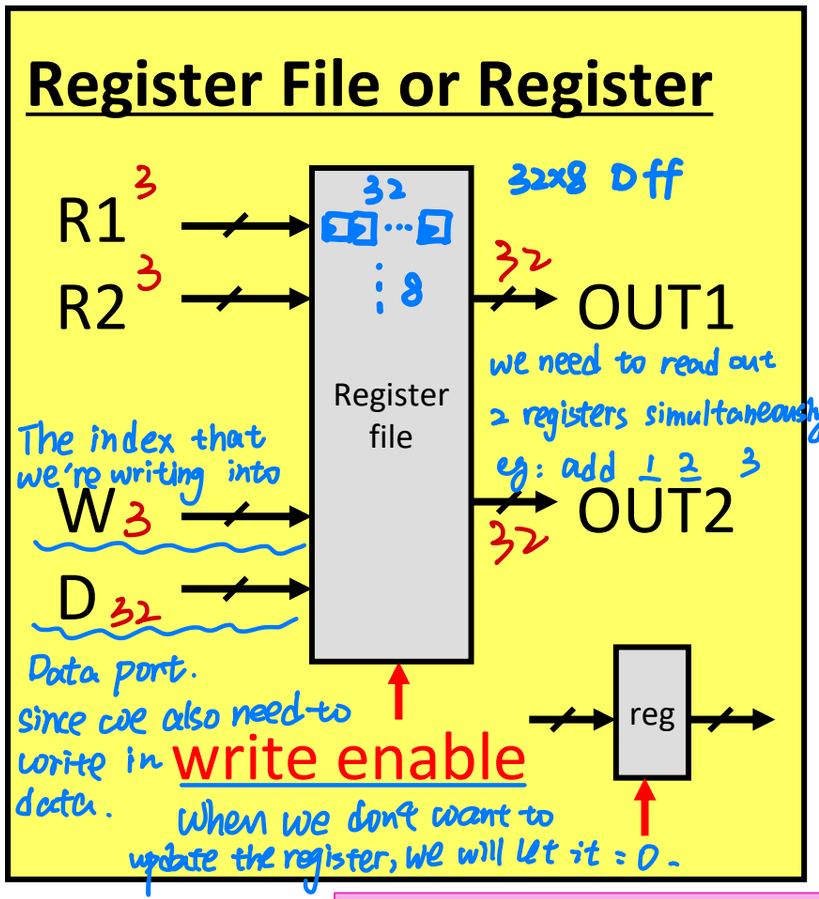
OUT(31:16) = IN(15)
OUT(15:0) = IN(15:0)

Useful when compute unit is wider than data



State Building Blocks (1)

hold all of the register in our program.



Small/fast memory to store temporary values

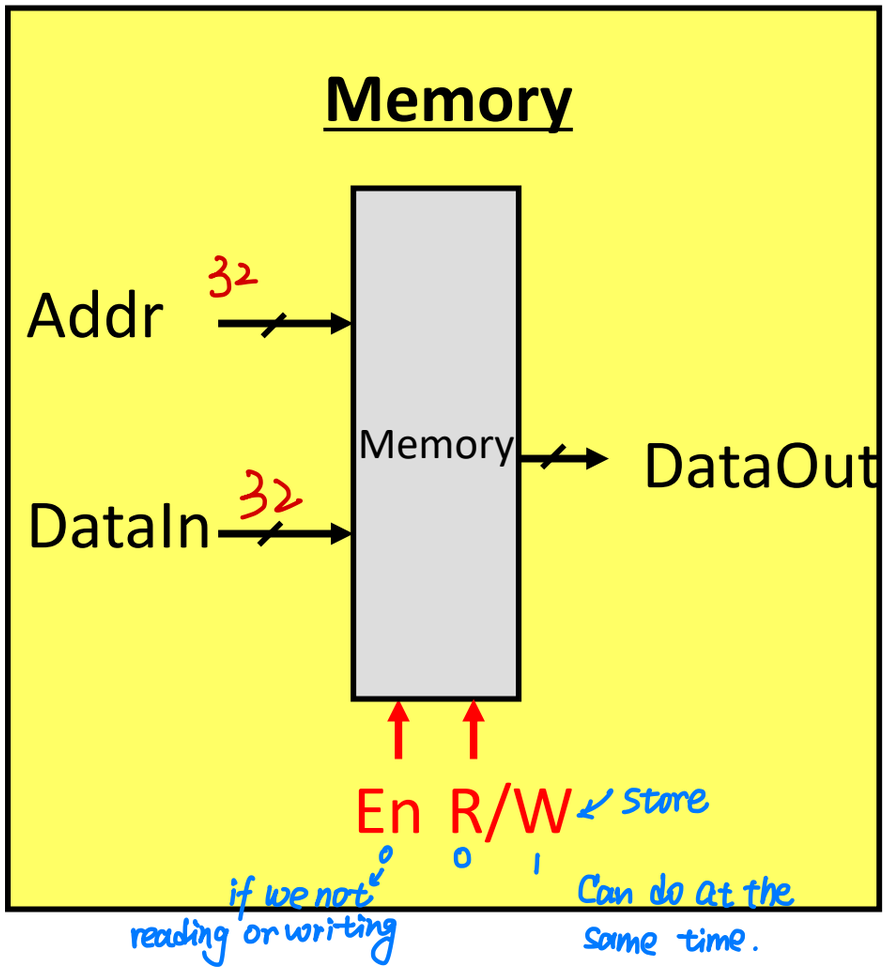
- **n** entries (LC2 = 8)
- **r** read ports (LC2 = 2)
- **w** write ports (LC2 = 1)

* Ri specifies register number to read
* W specifies register number to write
* D specifies data to write

Poll: How many bits are Ri and W in LC2K?

We have two different memories for our initial processor. ① holds instructions. ② holds just data.

State Building Blocks (2)



Slower storage structure to hold large amounts of stuff.

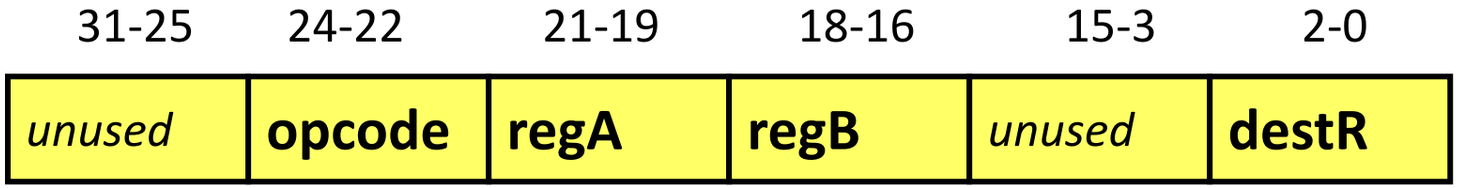
Use 2 memories for LC2K

* Instructions
* Data
* 65,536 total words

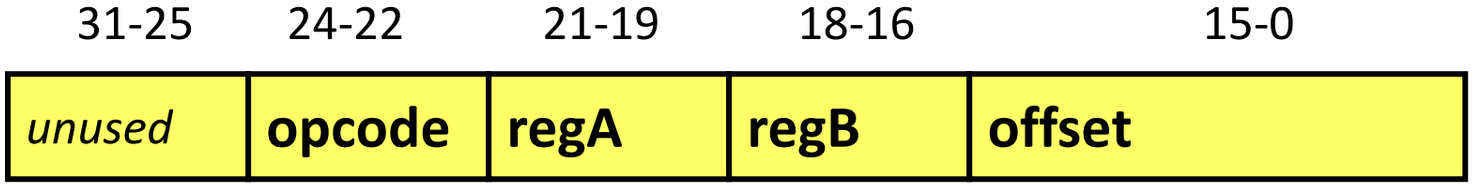
Recap: LC2K Instruction Formats

- Tells you which bit positions mean what

- R type instructions (add ‘000’, nor ‘001’)



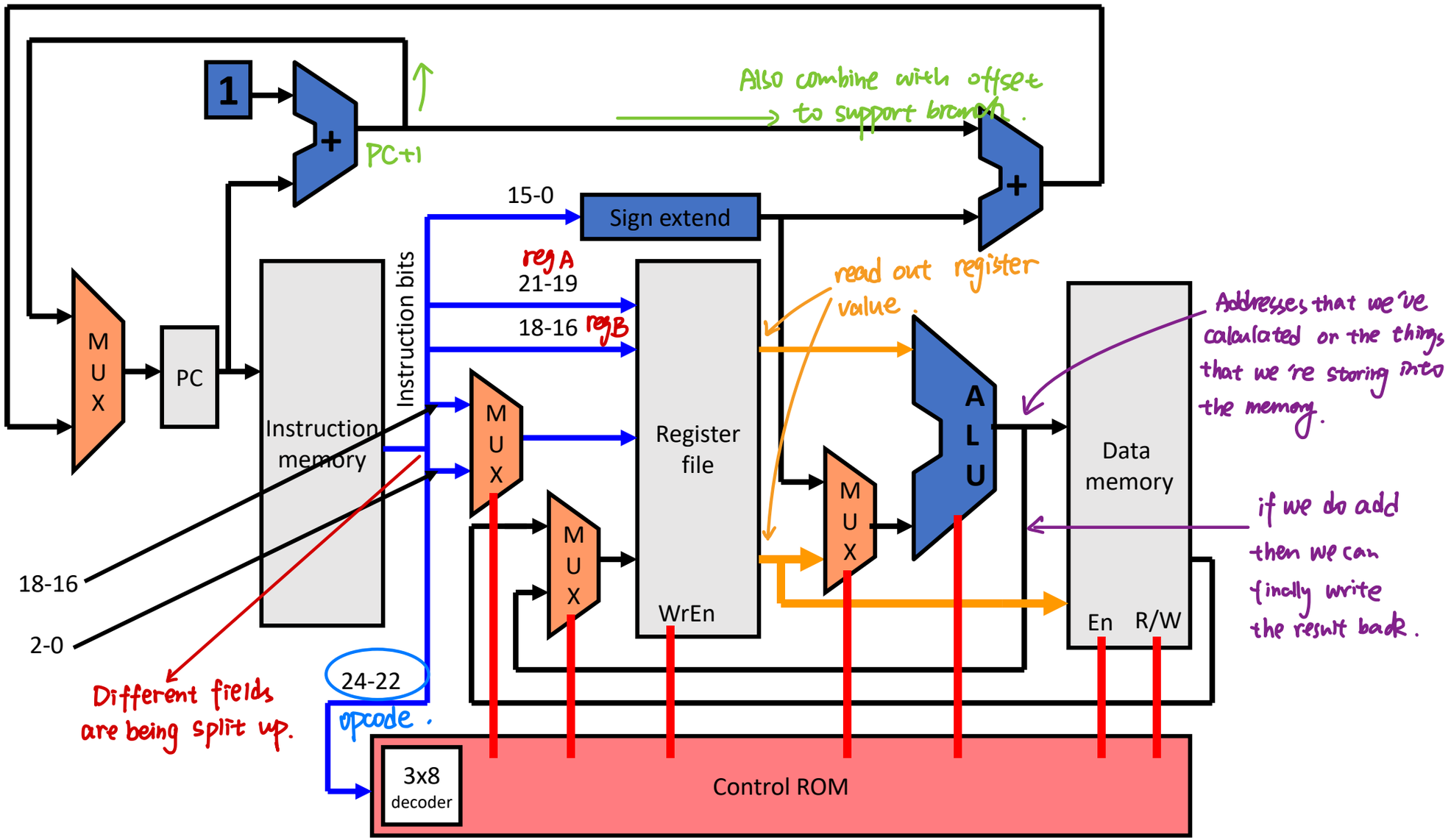
- I type instructions (lw ‘010’, sw ‘011’, beq ‘100’)



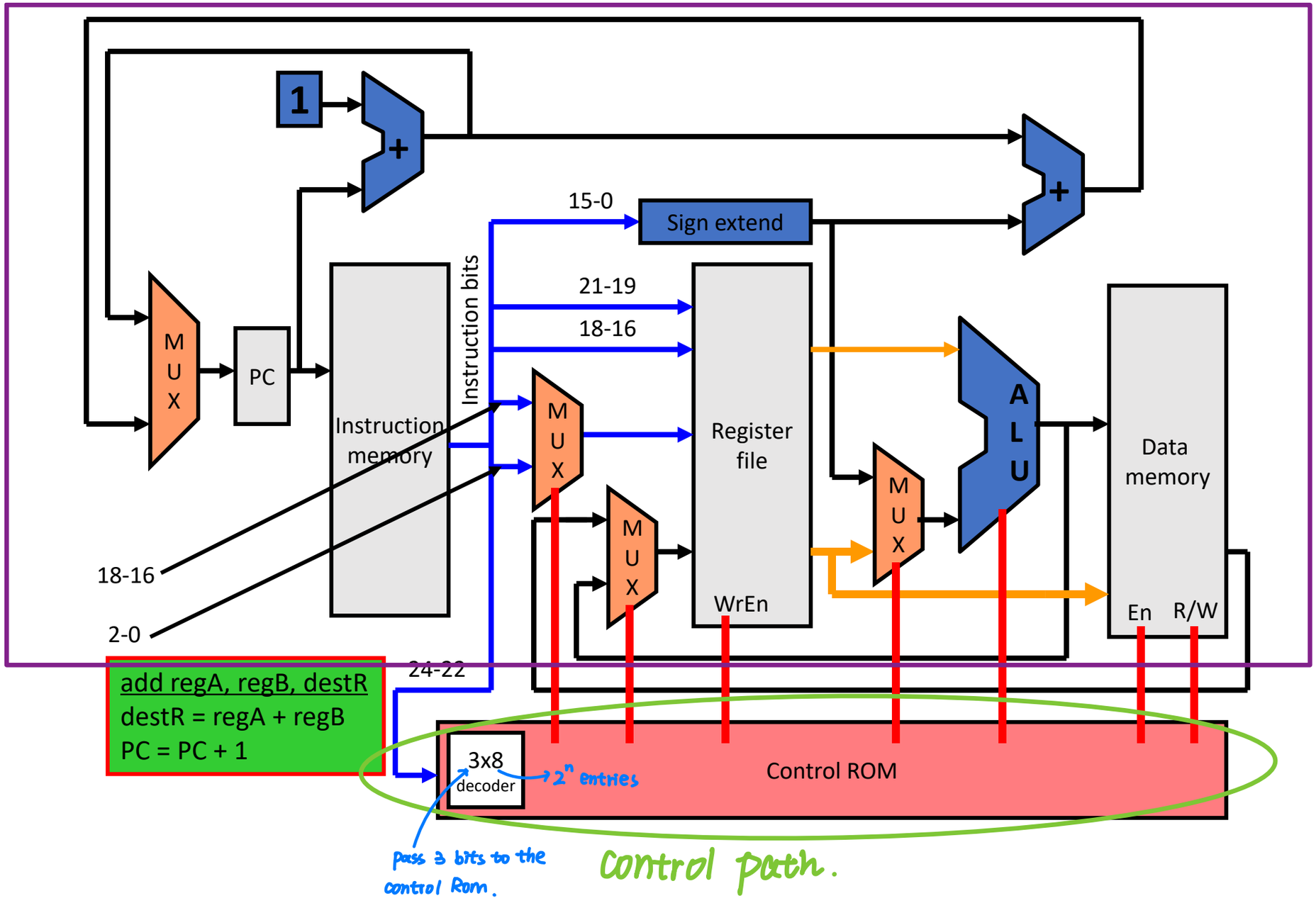
Agenda

- FSM Implementation
- ROMs
- Making our FSM more efficient
- Single Cycle Processor Design Overview
- **Supporting each instruction**
  - **ADD / NOR**
  - LW / SW
  - BEQ
  - JALR

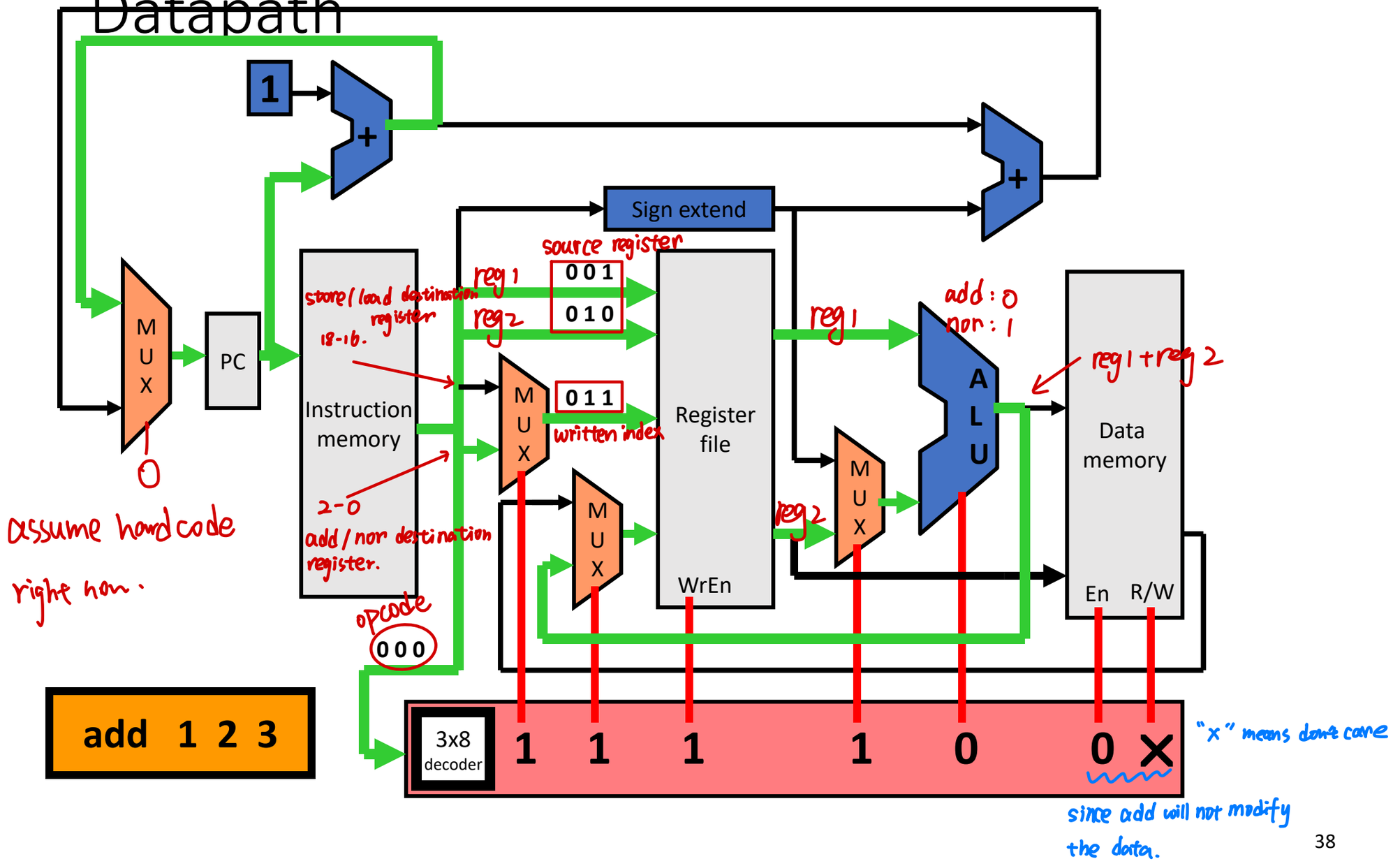
LC2K Datapath Implementation



Executing an ADD Instruction

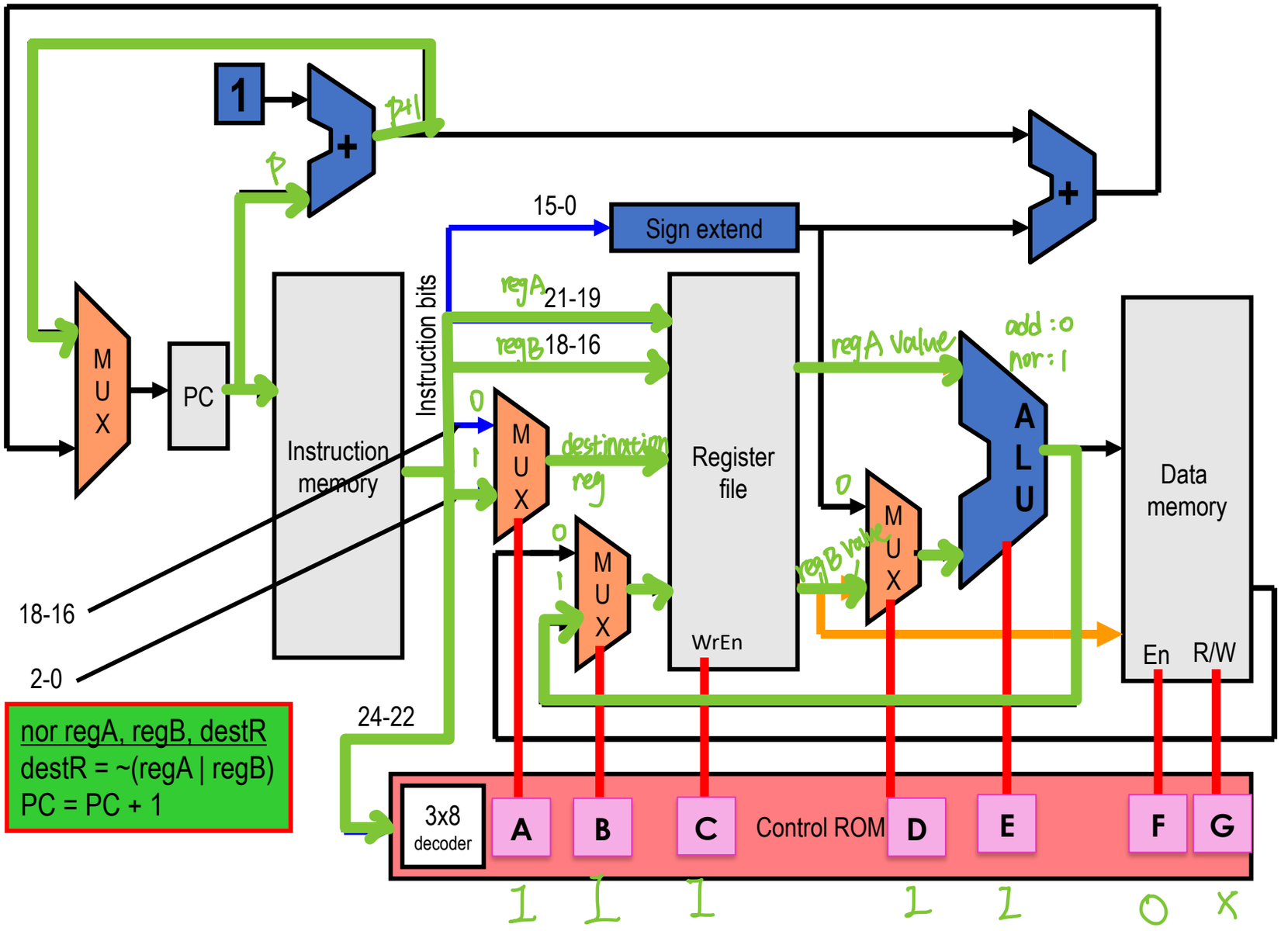


Executing an ADD Instruction on LC2K Datapath

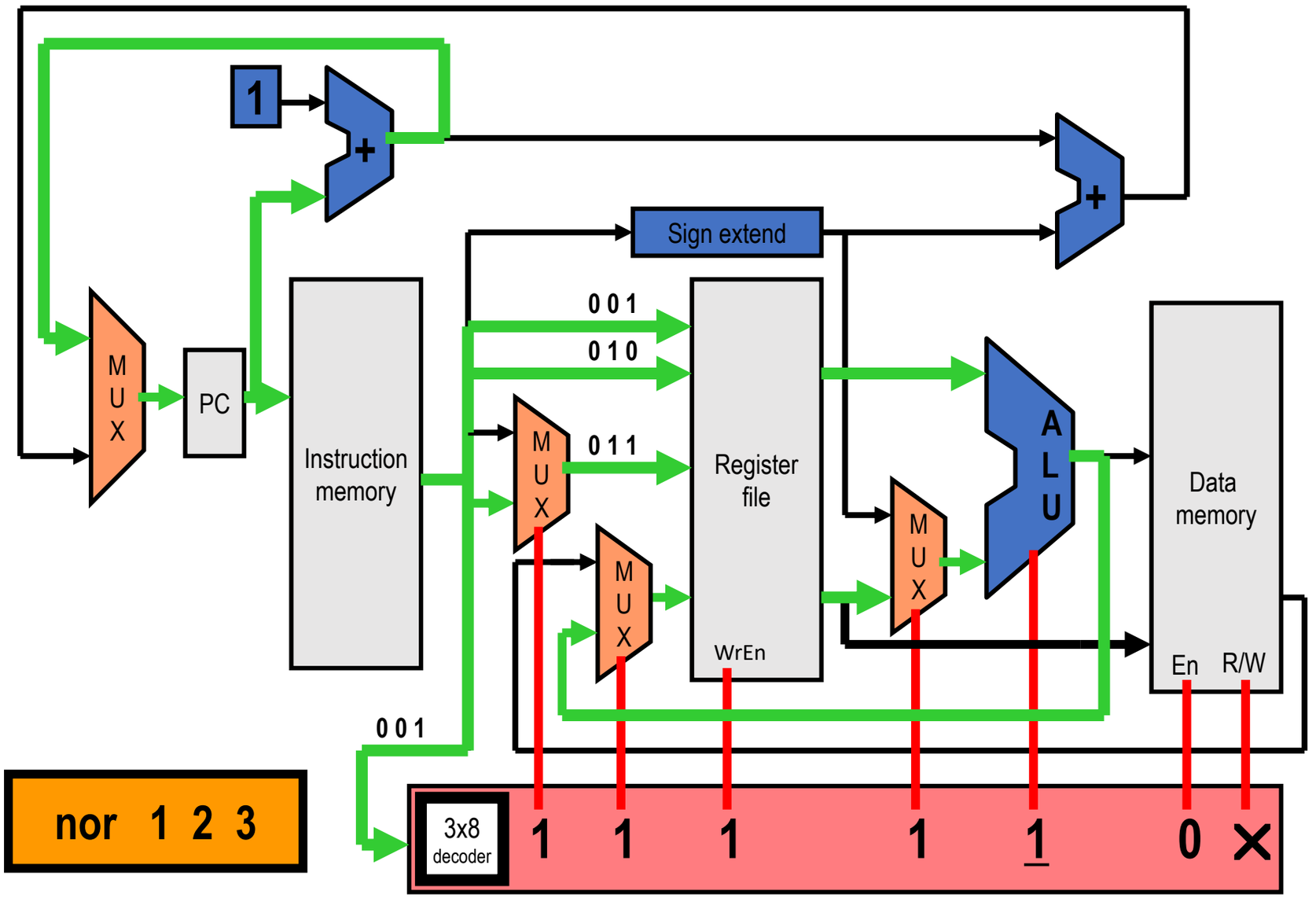


Executing a NOR Instruction

Poll: Which control bits need to be different from ADD?



Executing NOR Instruction on LC2K



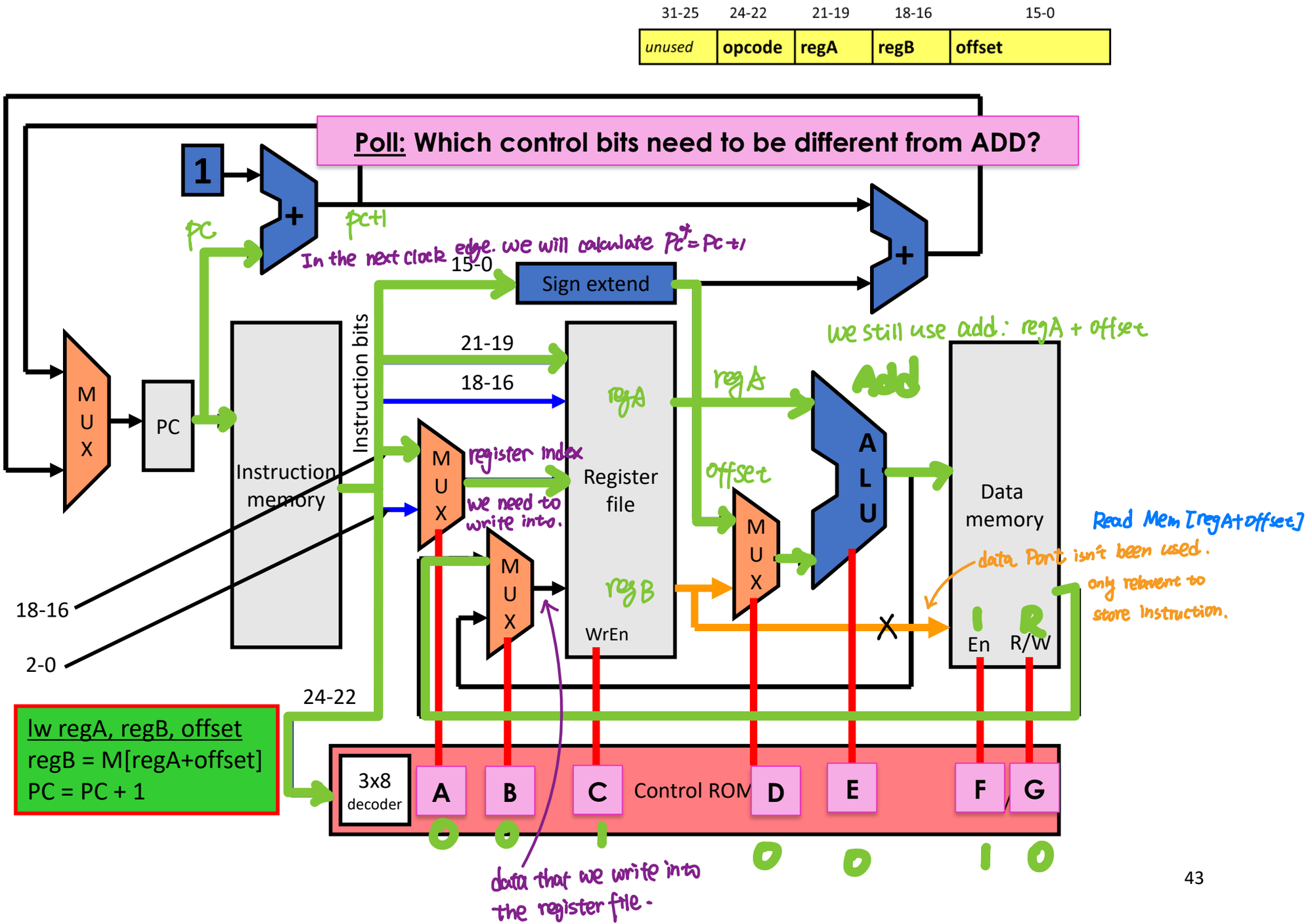
Next Time

- Finish up single-cycle and talk about multi-cycle

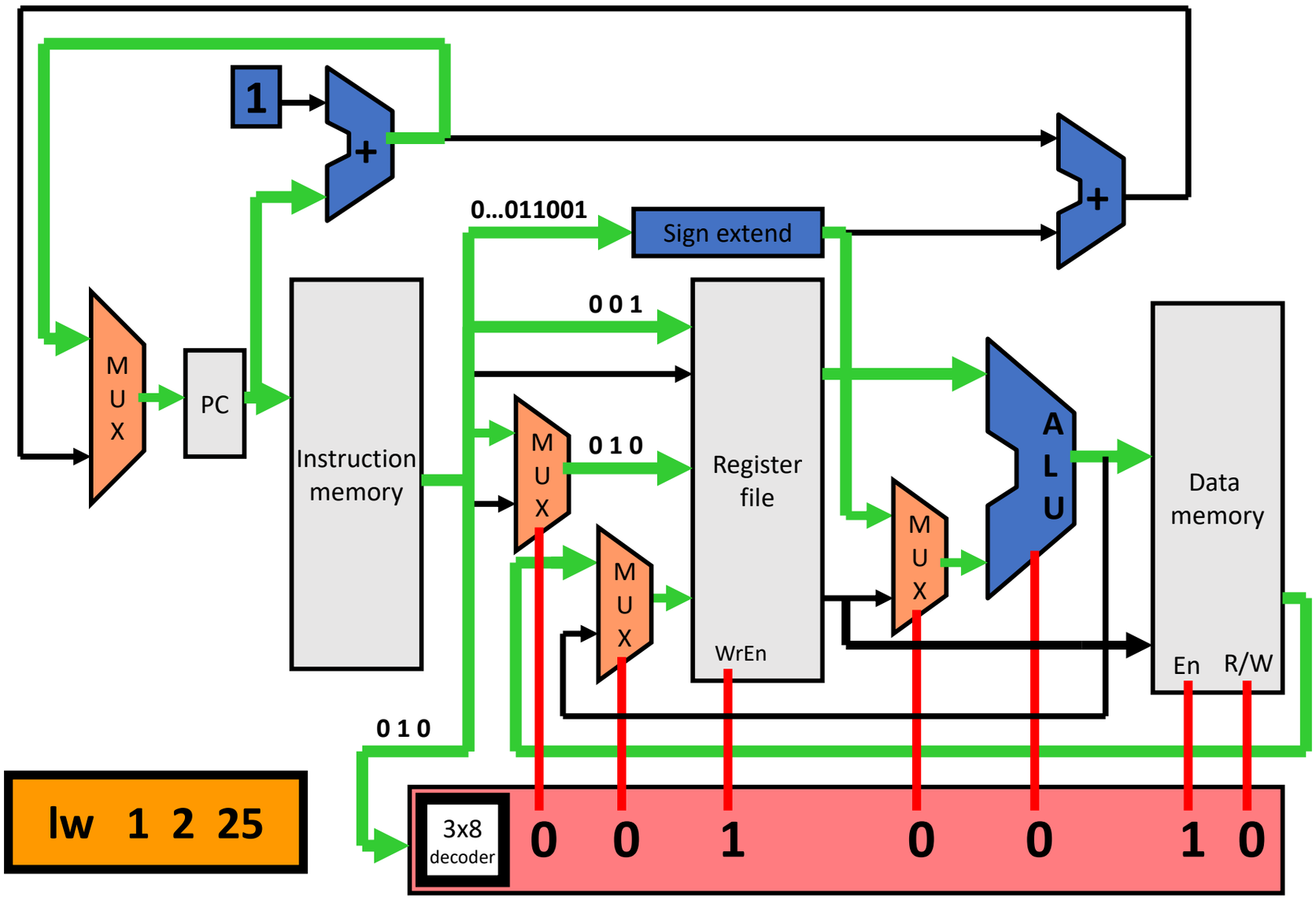
Agenda

- FSM Implementation
- ROMs
- Making our FSM more efficient
- Single Cycle Processor Design Overview
- Supporting each instruction
  - ADD / NOR
  - **LW / SW**
  - BEQ
  - JALR

Executing a LW Instruction



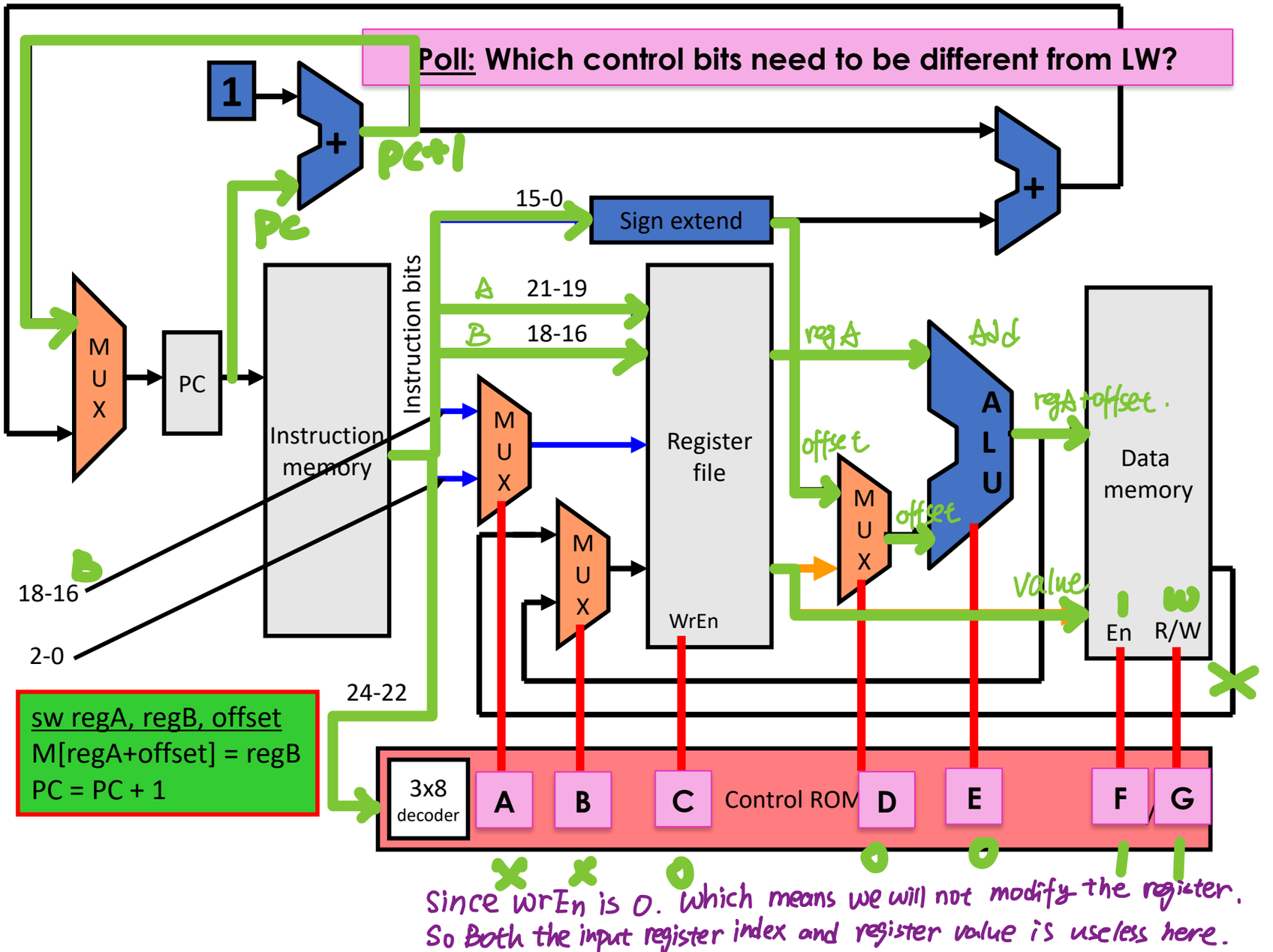
Executing a LW Instruction on LC2Kx Datapath



Executing a SW Instruction

opcode regA regB offset.

mem[regA + offset] = regB.



Executing a SW Instruction on LC2Kx Datapath